Wrocław, 4.04.2020 r

Michał Ryśkiewicz, 241383
Ocena:
Termin: WT/TN/7:30

Sprawozdanie z laboratorium nr 2 z przedmiotu

„Organizacja i Architektura Komputerów”

Rok akademicki 2019/2020, kierunek: INF

Prowadzący:

Mgr inż. Tomasz Serafin

1. Cel ćwiczenia

Zadaniami realizowanymi podczas zajęć było opracowanie zestawu funkcji realizujących operacje arytmetyczne takie jak: dodawanie, odejmowanie, mnożenie. Dodatkowo działania te miały być wykonywane na liczbach o długości przynajmniej kilkuset bitów, a także należało je wykonać w kodzie naturalnym i binarnym.

Głównym celem tych ćwiczeń było m. in. zapoznanie się z instrukcjami arytmetycznymi oferowanymi przez język asemblera, zrozumienie jak przechowywane są dane oraz czym są i jak działają poszczególne rejestry. Dodatkowo należało przypomnieć sobie informacje i podstawowe algorytmy działań arytmetycznych wyjaśnionych w ramach przedmiotu „Architektura Komputerów 1".

2. Przebieg pracy nad programem

Początkowym etapem pracy nad programem jakim należało się zająć było rozeznanie i poznanie mechanizmów w jaki sposób procesor dokonuje obliczeń. W tym celu konieczne było zapoznanie się do czego służą rejestry ogólnego przeznaczenia oraz rejestr flag. Dodatkowo istotnymi informacjami okazały m.in. reprezentacja danych w pamięci (wraz z reprezentacją kolejności, która w architekturze x86 występuje w formacie Little-endian) oraz tryby adresowania w języku asemblera.

Po przeprowadzeniu wstępnego rozeznania teoretycznego, pierwszym krokiem który wykonałem była rezerwacja odpowiedniego miejsca w pamięci na dane, oraz przypisanie im pewnych losowych wartości zapisanych w szesnastkowym systemie pozycyjnym. Następnie w sekcji „main" zaimplementowałem prostą pętle o zadanej długości, która pobierała 4 bajty zadanych liczb począwszy od najmłodszych bitów i wykonywała odpowiednie operacje arytmetyczne. Wynik operacji został umieszczany na stos lub do zmiennej, a odpowiednie przeniesienia były ustawiane w rejestrze flag. Po przekroczeniu odpowiedniej długości pętli, program przechodził do sekcji w której sprawdzane były ostatnie przeniesienia i umieszczane na stosie. W momencie kiedy wszystkie wyniki cząstkowe znajdywały się już na stercie, na jej szczyt umieszczany był tekst formatujący, a następnie wywoływana była funkcja „printf" w celu wyświetlenia wyniku na konsoli. Na koniec działania programu wykonane zostały odpowiednie wywołania systemowe, zakończone przerwaniem systemowym nr 0x80.

3. Napotkane problemy

Głównymi problemami jakie napotkałem w trakcie implementacji programu były m.in. sposób wyświetlania wyników na standardowe wyjście urządzenia oraz wyrównanie długości drugiej liczby jeżeli było to konieczne.

Pierwszym rozwiązanym przeze mnie problemem była kwestia wyświetlania odpowiednio wyników z danych algorytmów. Zastosowane przeze mnie rozwiązanie wymagało nabycia dodatkowej wiedzy teoretycznej, ponieważ skorzystałem z funkcji printf zawartej w języku C. Do tego posłużyłem się kompilatorem języka C – gcc będącym podstawowym kompilatorem w systemach uniksowych. Dodatkowo dowiedziałem się w jakiej kolejności należy umieszczać dane na stos tak, aby funkcja printf odpowiednio je przetworzyła.

Drugą z kolei rzeczą, którą należało się zająć było odpowiednie wyrównanie długości zadanej liczby. Zgodnie z wytycznymi należało unikać operowania na stałych długościach liczb, dlatego rozwiązanie jakie zastosowałem opiera się o zarezerwowanie większej pamięci, np. 64 bajtów. W momencie tworzenia programu przestrzeń ta uzupełniana jest zerami, co pozwala mi następnie na kopiowanie poszczególnych segmentów krótszej liczby i umieszczaniem ich w zarezerwowanej przestrzeni.

4. Kluczowe fragmenty kodu

Poniżej prezentuję kluczowe procedury znajdujące się w części programów.

1. Procedura wyrównywania liczby do zadanej długości.

```
mov $liczba2_len, %edi          #edi = rozmiar 2 liczby
dec %edi                        #edi--
equalNumbers:

  mov liczba2(,%edi,4), %eax    #skopiuj  fragment wartosci liczby 2 do eax
  mov bufor2(,%edi,4), %ebx     #skopiuj zera z bufora - domyslnie ustawiany na zera przez gcc

  adc %eax, %ebx                #dodaj wartosc

  mov %ebx, bufor2(,%edi,4)     #wynik dodawania dodaj do bufora

  cmp $0, %edi                  #sprawdzenie czy przeiterowalismy po dlugosci calej liczby
  jz process

  sub $1, %edi

jmp equalNumbers
```

2. Procedura wykonywania algorytmu na przykładzie algorytmu dodawania

```
process:
xor %edi, %edi                  #zerowanie rejestru
mov $liczba1_len, %edi          #4 "segmenty" liczby1 -- 4 iteracje
clc                             #CF = 0
pushf                           #PUSH Flag register onto stack -- CF Flag

addAlgo:
  clc                           #Zerujemy przeniesienie
  popf                          #Pop Stack into flag

  mov liczba1(,%edi,4), %eax    #Przesunięcie(baza,indeks,mnożnik) baza i indeks -rejestry
  mov bufor2(,%edi,4), %ebx

  adc %eax, %ebx                #Dodaj wartości w rejestrach a i b

  push %ebx                     #Wynik dodawania wrzuć na stos

  pushf                         #I zapisz flagę przeniesienia

  cmp $0, %edi                  #licznik -- jeśli rejestr %edi == 0 to...
  jz lastCarry                  #...skacz do przeniesienia
  sub $1, %edi                  #Jeśli cmp != 0 to przejdzie tutaj i odejmie od licznika 1
  jmp addAlgo                   #Przeskocz na początek pętli
```

3. Procedura ostatniego przeniesienia na przykładzie algorytmu dodawania

```
lastCarry:
  popf                      #Ściągnij flagę przeniesienia ze stosu

  xor %eax, %eax
  xor %ebx, %ebx            #Wyczyść rejestry 2 wersja.

  adc %eax, %ebx            #Dodaj Z UWZGLĘDNIENIEM FLAGI CF

  push %ebx                 Wynik dodaj na stos
```

4. Procedura wyświetlająca dane na standardowe wyjście.

```
end:
  push $printer     #dodaj na stos format wyświetlania danych
  call printf       #wywołaj funkcję printf do wyświetlenia wyników
```

5. Opis uruchomienia programu

Do uruchomienia programów wykorzystałem komendy podane w bashu w postaci:

1. „gcc -m32 -ggdb adder.s”
2. „./a.out”
3. Ewentualne do debugowania „gdb a.out”.

Pierwsza z komend służyła stworzeniu programu przy użyciu kompilatora języka C. „gcc” odnosi się do samego kompilatora. „-m32” określa by dane wyjściowe były zapisane w formacie 32 bitowym. „-ggdb” nakazuje użycie debuggera, natomiast „adder.s” jest to nazwa pliku w którym znajduję się kod napisany w języku assemblera.

Druga komenda odnośni się do uruchomienia skompilowanego programu, którego domyślna nazwa to „a.out”.

Trzecia komenda odnosiła się do uruchomienia debuggera w poszukiwaniu ewentualnych błędów w trakcie tworzenia programów.

Zawartość pliku „makefile” dla programu dodającego liczby prezentuję poniżej.

```
all: adder

adder: a.out
        gcc -m32 -ggdb adder.s
```

6. Listingi zadań

1. Program dodający dwie liczby o dowolnej długości

```
.code32
SYSCALL = 0x80
SYSEXIT = 1
EXIT_SUCCESS = 0
SYSWRITE = 4
STDOUT = 1
SYSREAD = 3
STDIN = 0

.global main

.text
msg: .ascii "Podane liczby to: \n"
msg_len = . -msg
printer: .ascii "Wynik to: %u%u%u%u%u \n"

.data
bufor2: .space 64,  # zarezerwowane miejsce w pamięci na drugą liczbę z uwzględnieniem wiodących zer
liczba1:
.long 0x11111111, 0x11111111, 0x11111111, 0x11111112, 0x12345678, 0x12345678, 0x12345678,
0x12345678
liczba1_len = (. - liczba1)/4                    # dlugosc liczby

liczba2:
.long 0x11111111, 0x11111111, 0x11111111, 0x11111111, 0x11111111

liczba2_len = (. - liczba2)/4

main:

mov $liczba2_len, %edi          # edi = rozmiar 2 liczby
dec %edi                        # edi--
equalNumbers:

  mov liczba2(,%edi,4), %eax     # skopiuj  fragment wartosci liczby 2 do eax
  mov bufor2(,%edi,4), %ebx     # skopiuj zera z bufora -- domyslnie ustawiany na zera przez gcc

  adc %eax, %ebx                # dodaj wartosc

  mov %ebx, bufor2(,%edi,4)     # wynik dodawania dodaj do bufora

  cmp $0, %edi                   # sprawdzenie czy przeiterowalismy po dlugosci calej liczby
  jz process

  sub $1, %edi


jmp equalNumbers

process:
xor %edi, %edi                   #zerowanie rejestru
mov $liczba1_len, %edi          #4 "segmenty" liczby1 -- 4 iteracje
clc                             #CF = 0
pushf                           #PUSH Flag register onto stack -- CF Flag
```

```
addAlgo:
  clc                     #Zerujemy przeniesienie
  popf                    #Pop Stack into flag

  mov liczba1(,%edi,4), %eax     #Przesunięcie(baza,indeks,mnożnik) baza i indeks -rejestry
  mov bufor2(,%edi,4), %ebx

  adc %eax, %ebx              #Dodaj wartości w rejestrach a i b

  push %ebx                   #Wynik dodawania wrzuć na stos

  pushf                     #I zapisz flagę przeniesienia

  cmp $0, %edi              #licznik -- jeśli rejestr %edi == 0 to...
  jz lastCarry                #...skacz do przeniesienia

  sub $1, %edi                #Jeśli cmp != 0 to przejdzie tutaj i odejmie od licznika 1
  jmp addAlgo                 #Przeskocz na początek pętli


lastCarry:

  popf                      #Ściągnij flagę przeniesienia ze stosu

  xor %eax, %eax
  xor %ebx, %ebx              #Wyczyść rejestry 2 wersja.

  adc %eax, %ebx              #Dodaj Z UWZGLĘDNIENIEM FLAGI CF

  push %ebx                   #Wynik dodaj na stos

  jmp end

  end:
  push $printer               #dodaj na stos format wyświetlania danych
  call printf               #wywołaj funkcję printf do wyświetlenia wyników



exit:
  mov $SYSEXIT, %eax
  mov $EXIT_SUCCESS, %ebx
  int $SYSCALL
```

2. Program odejmujący liczby dowolnej długości

```
.code32
SYSCALL = 0x80
SYSEXIT = 1
EXIT_SUCCESS = 0
STDOUT = 1
SYSWRITE = 4
SYSREAD = 3
STDIN = 0

.global main

.text
msg: .ascii "Wynik to: \n"
msg_len = .- msg
printer: .ascii "Wynik to: %u%u%u%u \n"

.data
odjemna:
 .long 0x98765432, 0x98765432, 0x98765432, 0x98765432
odjemna_len = (. - odjemna)/4

odjemnik:
.long 0x11111111, 0x11111111, 0x11111111, 0x11111110
odjemnik_len = (. - odjemnik)/4

main:

   xor %edi, %edi
   mov $odjemnik_len, %edi

   clc
   pushf

substractAlgo:
   clc
   popf

   mov odjemnik(,%edi,4), %eax
   mov odjemna(,%edi,4), %ebx

   sbb %eax, %ebx
   push %ebx
   pushf

   cmp $0, %edi
   jz lastCarry

   dec %edi
   jmp substractAlgo

lastCarry:

   popf
   xor %eax, %eax
   xor %ebx, %ebx
   sbb %eax, %ebx
   push %ebx
   jmp end
```

```
end:
   push $printer
   call printf

exit:

   mov $SYSEXIT, %eax
   mov $EXIT_SUCCESS, %ebx
   int $SYSCALL
```

3. Program mnożący liczby dowolnej długości

```
.code32
SYSCALL = 0x80
SYSEXIT = 1
EXIT_SUCCESS = 0
SYSWRITE = 4
STDOUT = 1
SYSREAD = 3
STDIN = 0

.global main

.text
msg: .ascii "Podane liczby to: \n"
msg_len = . -msg
printer: .ascii "Wynik to: %u%u%u%u%u \n"

.data
bufor2: .space 64,  # zarezerwowane miejsce w pamięci na drugą liczbę z uwzględnieniem wiodących zer

liczba1:
.long 0x11111111, 0x11111111
liczba1_len = (. - liczba1)/4                    # dlugosc liczby

liczba2:
.long 0x00000000, 0x00000022
liczba2_len = (. - liczba2)/4

wynik: .space 64,
wynik_len = (. - wynik)/4

wynik_offset: .long 0x4

main:
mov $liczba1_len, %edi
mov $liczba2_len, %esi
dec %esi
push $0

firstloop:
mov $liczba1_len, %edi          # 4 "segmenty" liczby1 -- 4 iteracje
dec %edi
clc                              # CF = 0
pushf                           # PUSH Flag register onto stack -- CF Flag
```

```
multAlgo:
  clc                    # Zerujemy przeniesienie
  popf                    # Pop Stack into flag

  mov liczba1(,%edi,4), %eax     # Przesunięcie(baza,indeks,mnożnik) baza i indeks -rejestry
  mov liczba2(,%esi,4), %edx

  imul %edx           # Dodaj wartości w rejestrach a i b

  pop %ebx
  add %ebx, %eax

  mov $wynik_offset, %ecx
  mov (%ecx), %ecx
  pushf
  mov $liczba1_len, %ebx

  sub %ebx, %ecx
  add %edi, %ecx
  inc %ecx
  popf

  mov %eax, wynik(,%ecx ,4)              # Wynik dodawania wrzuć na stos
  push %edx

  pushf                                  # I zapisz flagę przeniesienia

  cmp $0, %edi                           # licznik -- jeśli rejestr %edi == 0 to...
  jz carry                               # ...skacz do przeniesienia

  sub $1, %edi                           # Jeśli cmp != 0 to przejdzie tutaj i odejmie od licznika 1
  jmp multAlgo                           # Przeskocz na początek pętli

carry:
  popf                                   # Ściągnij flagę przeniesienia ze stosu
  xor %eax, %eax
  xor %ebx, %ebx                         # Wyczyść rejestry 2 wersja.
  pop %edx
  dec %ecx
  adc %edx, wynik(,%ecx ,4)

  mov $wynik_offset, %eax
  mov (%eax), %eax
  dec %eax

  mov $1, %ebx
  mov %eax, wynik_offset(,%ebx,4)
  dec %esi
  cmp $0, %esi

  jle temporary

  jmp firstloop


temporary:
  mov $wynik_len, %eax
```

```
end:
  mov wynik(,%eax, 4), %ebx
  push %ebx
  dec %eax
  cmp $0, %eax
  jg end

  push $printer              # dodaj na stos format wyświetlania danych
  call printf                # wywołaj funkcję printf do wyświetlenia wyników

exit:
  mov $SYSEXIT, %eax
  mov $EXIT_SUCCESS, %ebx
  int $SYSCALL
```